Vulnerable, or unsafe functions
strcpy does not check input size
strcpy(buf, str) simply copies memory contents into
buf starting from *str until “0” is encountered,
ignoring the size of area allocated to buf

Many C library functions are unsafe
strcpy(char *dest, const char *src) — copy from src to dest
strcat(char *dest, const char *src) — appends the src string to dest
  ● E.g., dest[126] = “abc” , strcat(dest, “def”) → dest = [abcdef]
gets(char *s)
printf(const char *format, …)

strncpy and strncat are examples of “safer” versions.

strncpy:

strncpy(char *dest, const char *src, size_t n)
If strncpy is used instead of strcpy,
no more than n characters will be copied
from *src to *dest

IMPORTANT: Programmer has to supply the right value of n

apache 1.3 example (in worksheet with gaps)
Problem:
  * Put up to MAX_STRING_LEN-1 characters into buffer
  * add “:” to buffer
  * AGAIN put up to MAX_STRING_LEN-1 characters into buffer
  * Easy to see in retrospect, but highlights challenges in designing software securely

======================================================================
Off-By-One Overflow
(on worksheet with gaps)

1-byte overflow: can’t change RET, but can change
pointer to previous stack frame

On little-endian architecture, adversary can make it point into buffer.
RET for previous function will be read from buffer!
This is only possible if the buffer is within 256 of the original saved FP.

======================================================================
Format strings (e.g., %d, %X, etc. Indicates how to interpret a value from the stack)

Proper use:

```c
int foo=1234;
printf("foo = %d in decimal, %X in hex",foo,foo);
```

Improper use:

```c
char buf[14]="Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

Another example:

```c
proper
printf("Here is an int: %x",i);

what if:
char buf[16]="Here is an int: %x";
printf(buf);

what if:
char buf[16]="Here is a string: %s";
printf(buf);
```

If buffer contains format symbols starting with %, location pointed to by printf's internal stack pointer will be interpreted as an argument of printf. This can be exploited to move printf's internal stack pointer.

Example: buf = "str %s" → printf(buf); Stack: [...........][saved FP][ret IP][*buf][.....][............]
- printf will interpret the region of memory adjacent to buf as a ptr to a string.

**DRAW the STACK**

The key to exploiting printf is %n!!!

%n format symbol tells printf to write the number of characters that have been printed

```c
printf("12345%n", &myVar); -- writes 5 into myVar.
```
Now what about

```c
char buf[16]="Overflow this!%n";
printf(buf);
```

Stack location pointed to by printf’s internal stack pointer will be interpreted as address into which the number of characters will be written.

Let’s say our buf is “……%n……%n……%n……%n”
Use multiple “%n”s to overwrite different bytes of the return address.

Other ways to manipulate format strings (e.g., “%-80x” outputs 80 characters right justified)

Also recall printf("%Md", 10) for M.

EXAMPLE:
func calls printf.
func has a local variable buf, which the attacker can completely control
call to printf uses buf as a parameter (so pointer to buf appears as parameter)
buf has % parameters to move printf stack to addresses of return address in buf
buf uses %Md for values of M to affect number of output characters
repeat for &RET, &RET+1, &RET+2, &RET+3.